Towards millions of communicating threads

Hoang-Vu Dang
Department of Computer Science
University of Illinois at Urbana-Champaign
hdang8@illinois.edu

Marc Snir
Department of Computer Science
University of Illinois at Urbana-Champaign
snir@illinois.edu

William Gropp
Department of Computer Science
University of Illinois at Urbana-Champaign
wgropp@illinois.edu

ABSTRACT

We explore in this paper the advantages that accrue from avoiding the use of wildcards in MPI. We show that, with this change, one can efficiently support millions of concurrently communicating light-weight threads using send-receive communication.

CCS Concepts

•Networks → Network algorithms; •Computing methodologies → Parallel algorithms; •Computer systems organization → Multicore architectures;

Keywords

Message Passing Interface, MPI, runtime system, communication, concurrent execution, multi-threading

1. INTRODUCTION

The first version of the Message Passing Interface (MPI) was designed more than twenty years ago [16]. The design was influenced by the existing message-passing formalisms, such as CSP [24], the practice of then extant message-passing libraries, and the needs of scalable parallel computers at the time. This resulted in fairly complex rules for matching sends to receives: MPI matches sends to receives using values for communicator, sender rank and tag; the receive operation can specify a wildcard value for sender rank and tag. Sends and receives must be matched in order: If a receive posted at A can match two distinct sends from B, then the receive will be paired with the older send; and if a send matches two distinct receives, it will be paired with the older receive. Finally, the implementation must handle sends that occur before a matching receive is posted, as well as those that occur in the reverse order.

Since communication was relatively slow, and the communication protocol was largely executed in software, the additional software overhead of a more complex protocol was not a significant issue. It also balanced against the advantage of supporting the practice of different message-passing libraries. Most importantly, MPI-1 targeted single threaded processes, since computers at the time used single core processors. MPI support for multithreaded processes appeared only in MPI-2 [20].

The situation has changed a lot in the last twenty years: Computer nodes can now have hundreds of concurrent hardware threads; modern adapters can support a significant fraction of the communication stack in hardware or firmware, while throughput-oriented cores execute the MPI library code more slowly. Programmers have shifted to the use of hybrid parallelism: Shared memory parallelism, e.g., with OpenMP for intranode, and message-passing parallelism for internode. However, MPI calls are executed only in sequential sections of the OpenMP code, and MPI is used in funneled mode, with all MPI calls being executed by one thread. Indeed, we could not find significant application codes with concurrent MPI calls. This constraint results in less efficient OpenMP code with more frequent serialization points.

One reason for this state of affairs is that the support for multithreaded MPI is still imperfect. The latest version of Open MPI still does not provide stable support of `MPI_THREAD_MULTIPLE` [44]. Several studies have reported significant overhead for performing MPI calls concurrently on many threads [22, 6]. This is, in part, an engineering issue: MPICH provided thread-safety via a coarse-grain lock which essentially serialized all MPI code. The research to replace coarse-grain locking is still ongoing [7, 6, 4], but progress is slow because the support for multithreading comes at the expense of single thread performance (which is what benchmarks measure). For example, there are a significant number of global objects shared by the executing threads. Global objects that implement stacks, queues or hash tables can be replaced with concurrent data structures, and others might be protected with finer-grain locks. However, finer grain locks and concurrent data structures have higher overheads that add to the critical path length and reduce the performance of single-threaded applications.

The semantics of matching sends to receives complicates the design of efficient support for `MPI_THREAD_MULTIPLE`. The simplest mechanism for matching sends to receives in the right order is to use a linked list for posted receives and a linked list for early arriving (unexpected) sends. When a message arrives, it is paired with the first matching receive in the posted receive list; if none is found, it is appended to unexpected send list. Symmetrically, when a receive is posted, it is paired with the first matching send in the un-

Euro MPI Sept 26th–28th, 2016
ACM ISBN 123-4567-24-567/08/06.
DOI: 10.475/123_4

expected send list; if none is found, it is appended to the posted receive list. With an increasing number of threads, there can be a larger number of concurrent sends or receives, hence longer lists are searched sequentially and concurrently updated; the updates must be atomic – leading to higher contention or more complex protocols.

Various attempts have been made to minimize this bottleneck by partitioning the range of communicator, sender and tag values (and using distinct lists for each partition). This is easy for communicators, but the use of wildcards complicates the approach: One either must insert wildcard receives in multiple queues, or maintain a separate queue for wildcard receives that needs to be searched concurrently with the regular one (see discussion in [18], where a partial solution is provided). An alternative approach is to relax the restriction of MPI semantics. For example, the MPI forum is currently discussing a proposal that would enable programmers to disable wildcards or relax ordering semantics, for some communicators [35].

Another impediment to the efficient support of `MPI_THREAD_MULTIPLE` is the lack of integration between the MPI library and the thread library. Consider a thread that executes a blocking MPI call, such as `MPI_RECV` or `MPI_WAIT`, the thread may yield and should be rescheduled when an appropriate communication completes. The MPI library can track, for each blocked thread, which communication event makes the thread runnable again. However, there is no simple way of passing this information to the scheduler. The thread library will wake up threads irrespective of the status of the communication they are waiting for; the thread will check some flag and yield again, if the communication is not complete. This is not a problem if most of the time only one thread is blocked, but can be very inefficient if the number of blocked threads is large.

MPI needs to evolve so as to support efficient concurrent communication from a large number of threads. This can be done by a combination of changes in the MPI implementation and relaxations of the MPI semantics. The evolution needs to be guided by a clear understanding of the trade-offs involved, hence this paper presents a first step in this direction. We study how efficiently we can support send-receive with more limited MPI send-receive semantics. This can be used both to provide a more efficient send-receive layer to applications that can accept the restrictions we impose, and to quantify the cost of supporting richer semantics. This bottom-up strategy provides insight on the minimal number of cache misses that are necessary in send-receive communication. The protocol we propose is simple enough to be implemented in hardware.

The main result of this paper is that, if wildcard receives are avoided, the performance difference between `MPI_THREAD_SINGLE` and `MPI_THREAD_MULTIPLE` largely disappears. Furthermore, communication performance does not deteriorate, even with many thousands of concurrently communicating threads.

The main ingredients of our efficient message-passing runtime are

- A light-weight thread scheduler using a bit-vector that requires a single write for marking a thread as runnable.
- A constant time overhead algorithm for MPI point-to-point communication utilizing a specially designed concurrent hash-table.
- A resource-aware and locality-aware concurrent memory pool for packet management.
- An MPI runtime design for scaling up to million communicating threads.

Our paper is organized as follows. The next section describes the simplified send-receive model that we target and introduces our runtime design. We present our implementation details and optimization in Section 3. Section 4 discusses experiment and results. Section 5 gives an overview of some related work. Section 6 addresses some the unanswered questions in our work. Finally Section 7 concludes our study and discusses future work.

2. RUNTIME ARCHITECTURE

2.1 Restricted Send-Receive Model

We focus in this paper on the exchange of messages that are sent from and received into contiguous buffers with matching size; the handling of datatypes would be done by a layer atop this basic communication layer. We assume that sends and receives are matched using a key k (in MPI, this is the `<communicator, sender, tag>` triplet), and do not support wildcards. Furthermore, we assume that each send can be matched by exactly one receive, so that ordering is a moot issue. We shall discuss in Section 6 how we can relax these assumptions; the only important one is the prohibition of wildcard receives.

We focus on performance-critical operations, namely sends and receives; the creation of communicators or of datatypes is not (yet) addressed. We also ignore, for the time being, one-sided operations.

We assume a light-weight thread library, where thread scheduling does not involve the kernel; the scheduler is not required to satisfy fairness conditions.

2.2 High-Level System Design

We propose an implementation of message-passing based on the following assumptions:

- Large number of concurrently communicating threads. Threads are lightweight; they are scheduled by a user-space scheduler that understands synchronization objects. The number of light-weight threads may be significantly higher than the number of physical threads, and over-decomposition may be used to hide communication latency.
- Large number of cores; it is possible to dedicate one or more cores to communication.
- The Network Interface Controller (NIC) can be accessed in user space; it has its own routing tables, in order to translate ranks in MPI_COMM_WORLD to physical node addresses; it has page tables, in order to translate virtual addresses to physical addresses. We consider in our work InfiniBand adapters, but the design should port to other adapters.
- We consider only x86_64 architeture; however, the technique is general enough to port to other architecture that supports atomic exchange and atomic bit manipulation.

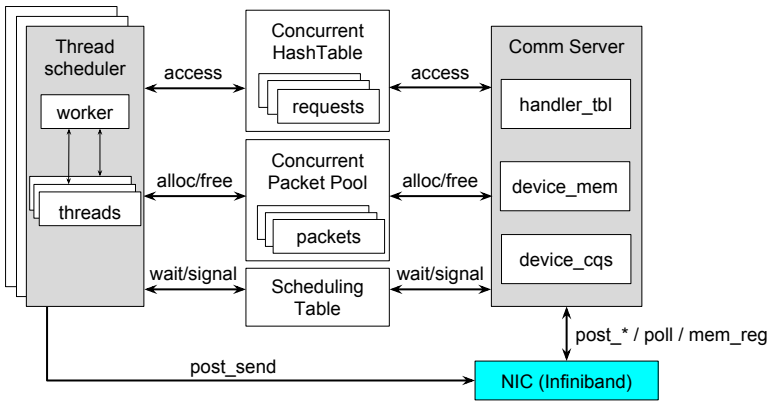


Figure 1: MPI Runtime Architecture for multi-threaded executions

Figure 1 shows the overall architecture of our described runtime system. We use a dedicated kernel thread as a *communication server*. This communication server executes all the communication protocol that is asynchronous w.r.t. the communicating *workers*, such as polling and handling the rendezvous protocol. Ideally, the communication server logic could be executed directly by the NIC. As the results in Section 4 show, one single-threaded communication server can support a large number of communicating worker threads. We plan to explore in future work the use of multithreaded communication servers.

The workers execute *User Level Threads* (ULT) (aka tasks). These are managed by a ULT scheduler. The scheduler is simply a function invoked when a ULT completes or yields. An MPI blocking call will block a ULT; the ULT scheduler will schedule another ULT on that worker. We assume in this paper that the association of ULTs to workers is fixed: ULTs are not migrated once they started executing. We discuss in Section 6 how one can remove this constraint.

The communication server and the workers share three data structures. Our simple design allows optimizations to be focused on these three critical shared data structures and operations on those:

- A *hash table* that is used to match sends and receives.
- A *scheduling table* that is used to mark which threads are runnable.
- A *packet pool* for packets posted to the NIC.

The hash table stores both unexpected incoming messages and outstanding receives. Since we assume there are no wildcards, we can hash by key (`<communicator, sender, tag>`). Furthermore, our assumptions imply that each communication will involve one insert in the hash-table, when no matching is found, and one delete, when a match is found. The insert is for the first occurring operation (either a receive or an unexpected send); the delete is for the second occurring operation of the send-receive pair.

The scheduling table is used by the communication server to mark a ULT as runnable when a communication operations it is waiting for has completed (it is also used for thread synchronization).

The packet pool supports two basic operations, namely `alloc` and `free`, which obtain and return a packet from/to the pool respectively.

2.3 Algorithms and Protocol details

The shared hash table H stores items that consist of a `<key,value>` pair $< k, v >$. The hash table supports one operation only, defined as follows

$$access(k,v) = \begin{cases} \textbf{if } < k,v' > \in H_{pre} \textbf{ then} \\ \quad H_{post} = H_{pre} - < k,v' >;\ \text{return}(v') \\ \textbf{otherwise} \\ \quad H_{post} = H_{pre} + < k,v >;\ \text{return}(\perp) \end{cases}$$

H_{pre} is the state of the hash table before the access and H_{post} denotes the state after the access.

In a concurrent setting, we require the hash table to be *linearizable* [23]. Informally, this means that the operations are atomic and appear to be executed at a point in time between the start and end of the operation. Linearizability is *composable* which allows us to correctly use the hash-table to implement other concurrent objects. In particular, this ensures that MPI calls take effect in program order.

Message delivery is implemented in two ways: *eager* or *rendezvous* protocol. Eager protocol is used for short messages: The message header and content are copied into a packet that is delivered to the network. The Send operation returns immediately as the send buffer can be reused. This protocol becomes inefficient when message size gets large. When this is the case, we switch to the rendezvous protocol in which the data is delivered directly from the source buffer to the target buffer by the NIC, thus saving extra copies. The rendezvous protocol requires additional messages to exchange control data and signal completion.

We describe below the eager protocol for blocking sends and receives. The protocol for nonblocking communication is similar, except that the receiving ULT may block (yield) when executing the Wait operation, rather than at the Send.

2.3.1 Eager protocol

The pseudocode for eager protocol is listed in Algorithm 1 and 2 for worker and communication server respectively. An eager send returns immediately, since the content of the

Algorithm 1 Eager-message send/recv for thread

```
1: procedure SEND-EAGER(b, s, k)          ▷ : buffer, size, key=<dest,tag>
2:     p = pkpool.alloc()
3:     Set packet header p.h to k
4:     Copy b to p.b
5:     Post p to network.
6: end procedure
7:
8: procedure RECV-EAGER(b, s, k)          ▷ : buffer, size, key =<from, tag>
9:     Create a request v = (b, s, t) ▷ buffer, size, thread id
10:    v' = H.access(k, v)
11:    if v' ≠ ⊥ then                     ▷ :match found
12:        Copy v'.p.b to b  ▷ : message arrived, copy data.
13:        pkpool.free(v'.p)
14:    else                               ▷ : insertion success
15:        ThreadWait()      ▷ : message not arrived, wait.
16:    end if
17: end procedure
```

Algorithm 2 Eager-message packet handler for communication server

```
1: procedure RECV-EAGER-PACKET(p)
2:     v' = H.access(p.k, p.v)
3:     if v' ≠ ⊥ then                     ▷ :match found.
4:         Copy p.b to v'.b  ▷ : message arrived, copy data.
5:         ThreadSignal(v'.t)             ▷ : mark receiver as runnable
6:         pkpool.free(p)
7:     else                               ▷ : insertion success.
8:         return  ▷ : message not arrived, nothing to do.
9:     end if
10: end procedure
```

data is copied over to a packet for transferring. Only the receiving algorithm needs some elaboration.

A pair of an arriving message and a matching receive causes two accesses to the hash table by the communication server; one for the arriving message and one by the worker thread for the posted receive. The first of these two operations inserts an entry in the table; the second deletes the entry, copies the data to the receive buffer and frees the packet. If the worker thread comes first, it will yield and will be marked runnable by the communication thread when the receive completes. If it comes second, it will complete the operation immediately.

Our matching mechanism requires to perform at most one operation on each of the three shared data structures. Therefore, the software communication overhead is bounded by a constant as long as these operations take constant time.

2.3.2 *Rendezvous protocol*

A rendezvous protocol usually involves the exchange of two control messages: a RTS (ready-to-send) issued by the sending worker thread, and RTR (ready-to-receive) issued by the receiver, after which data is sent. The data transfer can be supported efficiently using the Remote Direct Memory Access (RDMA) feature of modern NICs [26].

Since we do not support wildcards, RTS control message can be avoided: The receiver can send the RTR message when the receive is posted. The protocol is analogous to the eager-protocol, with roles reversed: The sender posts the RTS in the local hash table, and the communication server on the sender side posts the arriving RTR in that hash table. The second post results in a match and the RDMA transfer is initiated by the thread that made this second post. The communication servers on each side will mark the communicating ULTs as runnable when the RDMA completes. The RTR message carry buffer length information so that the sender can check for overflow.

3. RUNTIME IMPLEMENTATION AND OPTIMIZATION

3.1 Concurrent Hash-Table

We use a chained hash table with linked lists. The implementation is optimized for the limited usage we need.

Firstly, we can afford a *spinlock* per bucket, i.e., using an atomic boolean flag as a ticket to the critical section. This is a viable option since, given no collisions, there are at most two concurrent accesses per bucket by the communication server and a worker. Collisions can be minimized by matching the size of the hash table to the expected number of concurrent communications. A lock-free implementation (using Compare-And-Swap) is possible but it results in a more complex and time consuming code, thus we did not pursue this approach.

Secondly, in order to improve cache locality, we design each linked list element as a 4-entry array. Each entry consists of two 64-bit words. One of the entries is used as a *control entry* and the others are *data entry*. The control entry has the atomic flag for spin locking coupled with a 64-bit pointer to the next slot, in case one bucket contains more than three entries. A data entry consists of two 64-bit words of the key-value pair. The total size (64-bytes) thus typically fits in a cache line and one cache miss is the cost of both locking the bucket and fetching the data in the same cache line.

With the above optimization, the `access` operation expects one cache miss.

3.2 Thread scheduler

The thread scheduler needs to support the two operations `ThreadWait` and `ThreadSignal`. We designed a special thread scheduler (*Fult*) to optimize for these two operations. We compare this scheduler to the schedulers in the POSIX Thread library and in Argobots, a system that supports ULTs [39].

In Pthread and Argobots, the two operations are implemented using a `condition variable` with a Boolean flag, a generic container for many synchronization primitives. This typically requires a mutex and a queue to store waiting threads. An alternative is to use a busy-waiting synchronization flag that has lower latency; however this is not scalable since the processor spends useless time polling. Since Argobots implements ULT, its context-switching mechanism is similar to ours.

In Fult, we use a bit-vector to indicate runnable threads, instead of a queue structure. When a worker is created, it is assigned a unique worker id, denoted as ω. When a ULT starts running on the worker, it is also assigned an unique id Γ. A pair (ω, Γ) uniquely defines a ULT in the system at a

point in time. Since ULTs do not migrate, we can maintain a separate bit-vector for each worker; Γ is the index in the bit-vector structure for the bit indicating the status of the corresponding ULT (runnable vs. running/blocked).

Algorithm 3 further describes the bit-vector scheduler.

Algorithm 3 Thread scheduler using bit-vectors

```
1: procedure SCHEDULING(ω, V)        ▷ worker, bit-vectors
2:     while !ω.stop do              ▷ loop until user ask to stop
3:         for word in V do
4:             if word ≠ 0 then
5:                 localWord = 0
6:                 AtomicExchg(word, localWord)
7:                 while localWord ≠ 0 do
8:                     b =Leadingbit(localWord)
9:                     localWord = FlipBit(localWord, b)
10:                    ContextSwitch(b)
11:                end while
12:            end if
13:        end for
14:    end while
15: end procedure
```

In contrast to the 1-thread granularity of the queue structure, Fult scheduler works at 64-threads granularity. By using an atomic exchange instruction to swap the relevant word to a local variable, we are able to continuously perform read/write from/to this variable without accessing the main memory. An improvement in the instruction set (for example read-modify-write at the bit-level) could further improve this implementation.

A ULT executing a `ThreadWait` will invoke the scheduling code. The scheduler then looks for a new runnable ULT. `ThreadSignal` is a single atomic bit-set instruction (e.g., `lock bts` in x86). To facilitate more general functionality, we also implement a `ThreadYield`, which is simply a `ThreadWait` following by a `ThreadSignal` on self.

The implementation requires the worker to iterate over the bit-vectors word by word to find a runnable ULT. This is efficient for up to 512 threads per worker, since 8× 64-bit words still fit in a cache line. To support even more threads, we use a hierarchical bit-vector structure. More specifically, we use first level bit-vectors as "hint" to index into the second level bit-vectors. That is, each bit in the top-level bit-vectors indicates which bit-vector at the next level may have a bit on. A `ThreadSignal` performs a bit-set first into the lowest level then the higher one. On the other hand, the scheduler finds a thread by first looking at the top level then down to a lower level. The idea is similar to Bloom Filter technique [8], a query to the higher bit-vectors returns "possibly set" or "definitely not set" (or being set) for a lower bit-vectors.

If bit-vectors have s_1 bits at the top level and s_2 bits at the lower level, then the total number of threads we can support is $s_1 \times s_2$. We can choose $s_1 = s_2 = 512$ to fit each bit-vectors group in a cache line. Hence, a maximum of $256K$ concurrent ULTs per worker is supported.

Although fairness could be an issue, our scheduler maintains the property of progress. If a ULT is marked as runnable, it eventually will be scheduled in a bounded number of steps. To show that this is true, consider the atomic exchange as taking a snapshot of the global state. In this snapshot, if a ULT is marked, it will be scheduled after all ULTs having smaller index are scheduled, which is bounded by the total length of the bit-vectors.

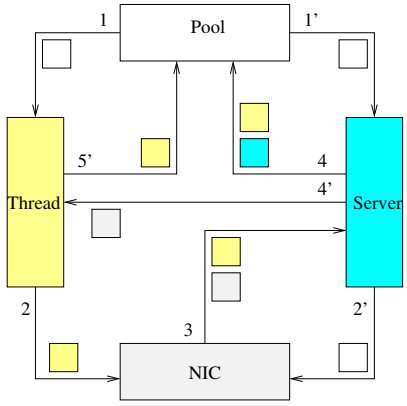


Figure 2: Packet life cycle. (1, 2) A worker sending data obtains a packet from the pool, fills the packet and submits to the NIC; (1', 2') Communication server obtains a packet for receiving data and posts the packet to the NIC; (3) The server polls the NIC and obtains the packet back (either sent (yellow) or received (gray) packet). (4) If the packet was for sending, it is returned to the pool (yellow); If the packet was for an incoming send and there is a match, the server copies the data and returns the packet to the pool (blue); (4') If the packet is for an incoming send and the there is no match, the packet is inserted to the hash-table (gray); (5') Following (4') - the worker obtains the packet from the hash-table and copies the data before returning it to the pool (yellow).

3.3 Concurrent Packet Pool

In general, the packet pool can be implemented using a lock-free stack. A pool `free` is translated to a stack `push`, and `alloc` is translated to a stack `pop`. At initialization, a fixed number of packets are initialized from the main memory and pushed onto the stack. The Last-In-First-Out (LIFO) property allows good temporal locality for writing/reading to/from the content of a data packet. In a single-threaded environment this design is sufficient for good performance, but not with multiple cores. Consider a packet recently used by a worker and returned to the pool: This packet could be subsequently obtained by a different worker running on a different core. This causes several cache misses since the cache line is alternately owned by each of the two workers. An example is when two threads running in two different cores alternatively perform `MPI_SEND`.

Figure 2 explains how different components in our system might change the affinity of data inside a packet. The figure shows the life of a packet from the time it leaves the pool until it is returned. As explained in the figure, when a packet returns to the pool, it was either last accessed by the communication server or by one of the workers. However, the affinity of a packet to a core is lost once it is posted to the NIC as a receiving buffer since it will be written by the NIC.

Analyzing the packet life cycle naturally motivates a new design for the packet pool: split the centralized packet management into a private pool per worker. Initially, there is a

fixed number of packets for each worker. At runtime, we allow moving packets among those pools via *resource-stealing*, similar to a non-blocking work-stealing algorithm [5].

A private pool is implemented as a fixed size double-ended queue (deque). The deque has three main operations: `popTop`, `pushTop`, and `popBottom` which allows LIFO accessing at one end and removing items from the other end. The packets at the bottom of the deque have been least recently used and are better candidates for use by threads other than the local worker. We use `popTop` for sending packets, and `popBottom` for receiving packets (used by the NIC) and for packet stealing (used by other workers). In case its private pool is empty, a thread *steals* packets from a randomly choosen private pool. Currently, we implement the pool using a simple ring-buffer and a spinlock.

We expect the pool operations to require at worst 3–5 memory accesses: lock, top, bottom and buffer pointers accesses; a memory read/write for storing a value into the container; and in the rare case of resource stealing, there could be more due to cache misses between processors

4. PERFORMANCE EVALUATION

4.1 Experimental Setup

All of our experiments are done on the Stampede cluster [2] at TACC. The cluster nodes are Intel Sandybridge x86_64 processors with Xeon Dual eight-core sockets, operating at 2.70 GHz, with 32 GB RAM. Each node is equipped with a MT4099 Infiniband FDR ConnectX interface that is capable of delivering 54 Gbps. The cluster runs MVAPICH2 MPI version 2.1, compiled with gcc version 4.9.1. All our codes were compiled with mpicc using gcc version 4.9.1 with `-O3` optimization. Unless noted otherwise, the configuration for MVAPICH2 is the default setting with shared memory optimization when running multiple MPI processes per node; when using with POSIX threads, `MPI_THREAD_MULTIPLE` and shared memory optimization are enabled. In both cases, thread affinity is also enabled.

4.2 Component overheads

In this section we evaluate our implementation of each individual component. Understanding them individually gives us an idea on the minimum cost of the overall system.

4.2.1 Concurrent Hash-Table

We measure the latency of hash-table operations in the two following scenarios, performing them with multiple POSIX threads:

- A thread performs an `access` when there is no item with the same key, which inserts the entry into the hash-table.
- A thread performs an `access` when there is already an item with that key, which deletes the entry from the hash-table.

To justify the benefit of customization, we compare performance to two popular general purposes hash-tables: libcuckoo implementing cuckoo hashing (*ch*) [29], and TBB concurrent hashmap (*tbb*) [37]. We ran the experiments 1000 times, each time a thread performs 256 operations. Between each run, we also perform a cache invalidation.

Figure 3 shows the result of our experiments for latency per operation. Both TBB concurrent hash map and libcuckoo show inconsistent latency when there are more concurrent threads, which is the result of conflicts. Our hash-table performs much better, with an overhead that is almost always as low as **0.05 usec**; the execution time has a low variance and is almost independent of the number of threads.

4.2.2 Thread scheduler

	POSIX thread	Argobots	Fult
Scheduling	0.75	0.08	0.02
Signal	1.15	0.30	0.01
Total	1.90	0.38	0.03

Table 1: Break down of thread scheduler overheads, shown in usec.

To evaluate the thread scheduler overhead, we measure separately the two operations: 1) How fast can a ULT be scheduled by performing a sequence of yields at a worker and 2) The cost of a `ThreadSignal` by repeatedly issuing the signal on a worker for a ULT at another worker. Table 1 shows our results averaged over 1000 runs. Our customized scheduler achieves a total cost of **0.3 usec**, for Signal+Yield, about 10× better than Argobots and 60× better than POSIX threads.

4.2.3 Concurrent Packet Pool

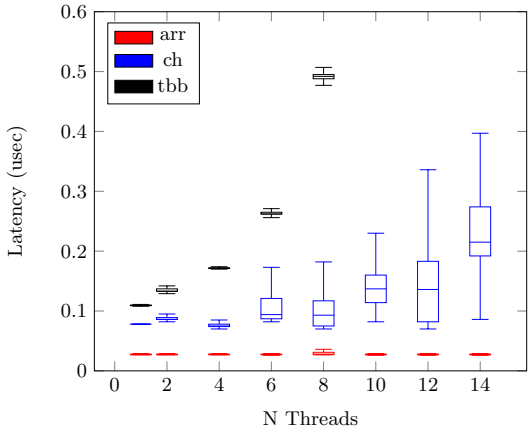
The overhead of packet pool operations is measured as the sum of the latency of an `alloc` and a `free` operation. We evaluate this quantity by performing a random number of `alloc` calls followed by the same number of `free` calls on each thread. To better match with a real workload, we also perform a random sleep in between the two groups of operations. The number of packets allocated per thread is always smaller than the total number of packets divided by the number of threads.

The result is shown in Figure 4, in comparison with implementation using a concurrent lock-free stack and a lock-free queue. Our result for this benchmark outperforms others by a wide margin, especially when the number of threads increases. The lock-free stack is faster than the queue for a single thread, however performs worse for more than two threads since there is contention at the top of the stack. The great variation in latency per operation of a centralized pool is clearly due to memory conflicts in this type of access patterns. Our latency is consistently in the range of **0.1 to 0.15 usec**.

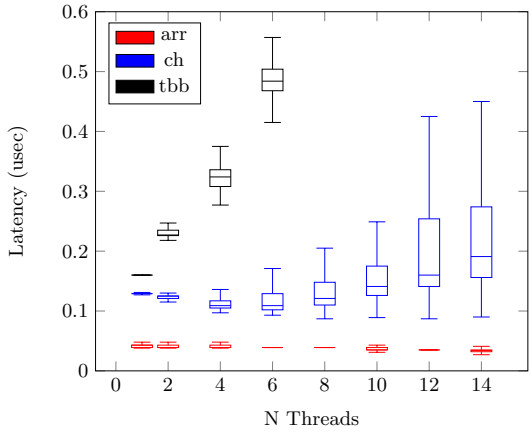
4.3 Microbenchmarks and Application Evaluation

In the previous section, we have analyzed our critical path by evaluating each individual component independently. Their latencies are added to the data movement overhead. In the tested system, the overall number that we achieve is **0.2–0.25 usec**. More importantly, we are able to maintain this performance with an increasing number of workers. In this section, we evaluate how they work together by using a set of microbenchmarks and applications.

Table 2 shows the different configurations that we evaluate. For MVAPICH2, we evaluate single threaded mode (*mvapich2*); single threaded mode with asynchronous progress



(a) Latency per successful `access`.



(b) Latency per failed `access`.

Figure 3: Latency of our hash-table implementation (arr) in comparison to libcuckoo (ch) and tbb concurrent hash map (tbb). Each hash table is created with the initial size of 2^{16}, the number of insertion per thread is chosen so that there is enough room and no expansion is required. TBB is also compiled with tbb-malloc to improve performance. Latencies exceeding 0.5 microseconds are not shown.

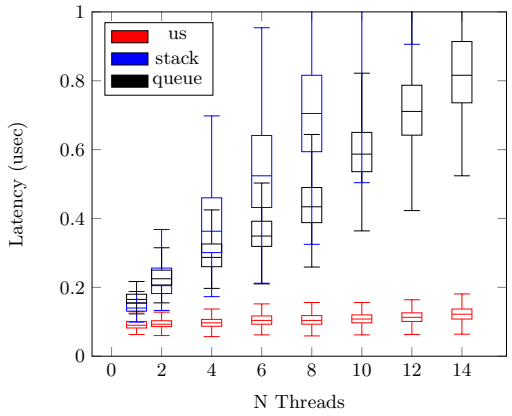


Figure 4: Latency of pool implementation vs. a lock-free pool and a lock-free queue implementation. Latencies higher than 1 microsecond are not shown.

	Msg Matching	Scheduler
mvapich2	queues	Single
mvapich2+async	queues	Single + Progress
mvapich2+mt	queues	POSIX thread
pthread+hash	hash-table	POSIX thread
abt+hash	hash-table	Argobots
fult+hash	hash-table	Fult

Table 2: Summary of MPI configurations used in the evaluation.

thread, i.e., `MPICH_ASYNC_PROGRESS`=1 (*mvapich2+async*); and multi-threaded mode (*mvapich2+mt*). Using an asynchronous progress thread reduces performance significantly for *mvapich2+mt* or when there are more processes per node, thus this option is not considered for those cases. For our implementation, we evaluate three different schedulers: POSIX Thread (*pthread+hash*), Argobots (*abt+hash*) and Fult (*fult+hash*).

4.3.1 OSU latency benchmarks

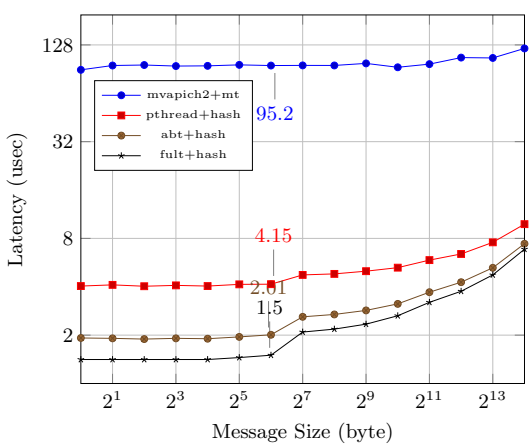
We use the OSU benchmarks [14] to evaluate the latency per `MPI_SEND` or `MPI_RECV`. The single threaded test is performed using `osu_latency`, the multi-threaded test is performed with `osu_latency_mt`. For a fair comparison in this experiment, we disable the MVAPICH2 *RDMA fast path* algorithms (by setting `MV2_USE_RDMA_FAST_PATH=0`) [30]. Further, in multi-threaded tests, we modify the code so that each thread uses different tags.

In order to evaluate the effect of cache conflicts between computation code and communication code, we also create another test by modifying `osu_latency` to add a number of random writes in between `MPI_RECV` and `MPI_SEND`. The random writes are uniformly distributed accross an array of $1GB$ size, thus eventually invalidate all the caches. The reported overhead is the overall time less the computation time measured when executing without communication.

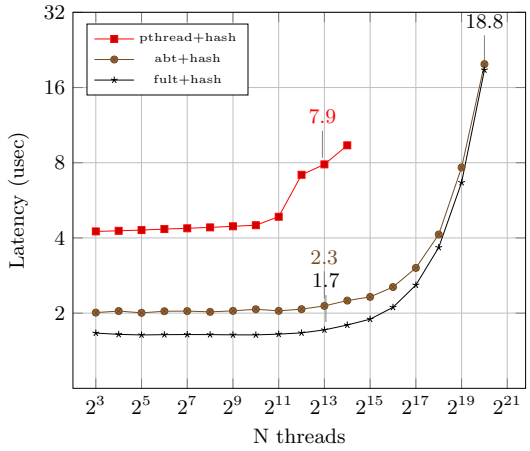
Performance results for the multi-threaded tests are shown in Figure 5. The largest improvement in performance is due to the replacement of the matching queues of MPI with the hash table. Then, the replacement of POSIX threads with light-weight threads. Our thread-scheduler improves the latency up to 40% compared to Argobots, 3× compared to POSIX thread scheduler. Overall, we achieve speedup of up to 60× compared to MVAPICH2. The typical communication overhead for MVAPICH2 with a single-threaded process is less than 2 usecs; with 8 threads, the overhead is close to 100 usecs due to synchronization overheads

Our scaling test in Figure 5(b) shows that we can support a very large number of threads with very small synchronization overheads. Our performance only degrades slowly at 16K communicating threads, which we attribute to the bottlenecks in memory for thread records (each thread is configured with a 16KB stack for this test).

Performance for the original OSU single-threaded test is shown in Figure 6(a). With our best implementation, we have lower latency than MVAPICH2 running with `MPI_-`



(a) Latency per message transfer for 8 threads, one per worker/core.



(b) Latency for 64-byte message transfer with up to 1M ULTs that are assigned round-robin to 14 worker/cores, *pthread+hash* version only works up to 16K threads.

Figure 5: Latency comparison between different MPI implementation using OSU multi-threaded latency test.

`THREAD_MULTIPLE` for small messages (as in *mvapich2+mt*) and virtually tie with MVAPICH2 running with `MPI_THREAD_SINGLE` (as in *mvapich2*).

Performance for the modified OSU single-threaded test is shown in Figure 6(b). At 2048 random writes, we can observe that the performance starts to be affected by cache effects; only the single-threaded MVAPICH2 still performs well. Nevertheless, our best implementation still outperforms MVAPICH2 under `MPI_THREAD_MULTIPLE`. Note that the slow-down can be due to a slower execution of the communication code as well as to a slower execution of the writes.

The comparison to MVAPICH2 is not entirely fair, since we do not support the long list of arguments and the many options for these arguments that MVAPICH has to support. However, the processing of the arguments of an MPI call is done independently by the calling thread; it does not result in synchronization overheads, and equally affect `MPI_THREAD_SINGLE` and `MPI_THREAD_MULTIPLE`. Our results indicate that we can support `MPI_THREAD_MULTIPLE` with a very small penalty, if any, compared to `MPI_THREAD_SINGLE`; and that still holds true even with 16K light-weight threads.

4.3.2 NAS Parallel Benchmarks - Data Traffic (DT)

The Data Traffic (DT) code is part of the NAS Parallel Benchmarks. It is used to evaluate the communication performance under three different communication patterns:

- Black Hole (BH): collects data from multiple sources to a single sink.
- White Hole (WH): distributes data from a single source to multiple sinks.
- Shuffle (SH): routes data from a small number of sources to a small number of sinks through a large numbers of layers.

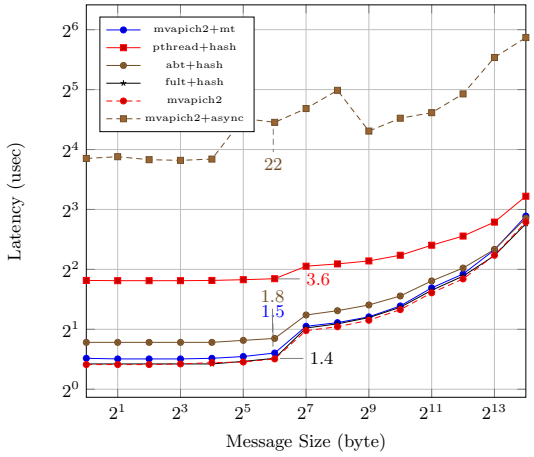
Between communication phases, there are also significant computations to verify results that help evaluate the ability to overlap communication and computation of the runtime system as well as the effect of cache locality. The application is written with MPI blocking send and receive and each destination rank has a uniquely assigned tag, making it a perfect use case for our MPI implementation. Hence, for this experiment, we execute the reference code using our MPI implementation without changing much of the source nor applying any threading. Since no threading is used, we run the benchmark on 128 nodes, one process per node, two cores per process. Our implementation uses a single worker in comparison with MVAPICH2 in sequential mode and MVAPICH2 with asynchronous progress.

The NBP suite also provides different classes of problem which represents different levels of scale. For the DT benchmark, we evaluate only class "A" since it is reasonably large (requires at least 80 processes), and moreover it is equipped with a proper verification. The reference code was downloaded from NBP suite version 3.3.1 at the NBP website [36].

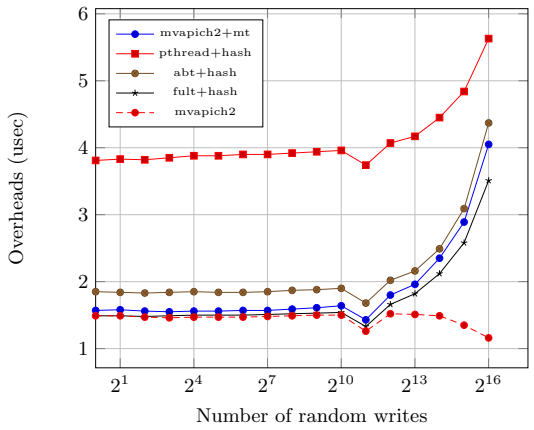
The results are shown in Figure 7. When there is a imbalance in the number of sources and sinks, we perform better in all cases, with up to 3× performance due to a better message matching algorithms. We are about 15% slower in the SH case, due to more cache conflicts. The *mvapich2+async* also reduces performance for the same reason, although since the network is polled from both the main thread and the helper threads, it is less effected (average L2 cache misses rate are 23%, 28% and 45% for *mvapich2*, *mvapich2+async* and *fult+hash* respectively as reported by the *perf* profiler).

4.3.3 Breadth-first-search (BFS)

BFS is the kernel for the *Graph500* benchmark [1], which is frequently used to determine the performance of supercomputers for latency bound applications. The MPI reference implementation generates a large-scale graph and assigns to each MPI process a fixed set of vertices. The implementation then has the processes cooperatively traverse the graph, starting from a particular vertex until all vertices are marked visited. Although the problem is simple, it is often



(a) Latency comparison for single-threaded OSU latency test.



(b) Latency comparison for 64-byte single-threaded OSU latency test with increasing number of writes.

Figure 6: Latency comparison for single-threaded OSU benchmarks

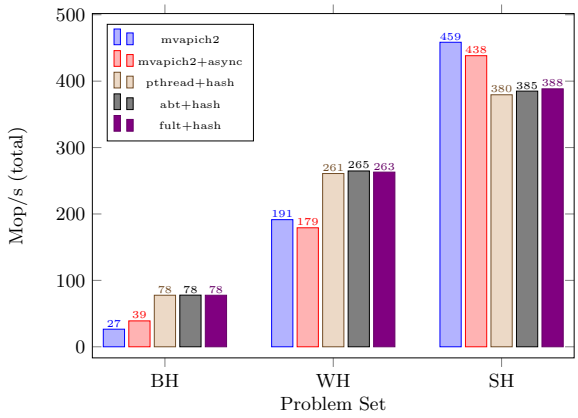


Figure 7: Performance of NAS-DT benchmarks in terms of million operations per second (Mop/s - the higher the better) for three different communication patterns under different MPI implementations.

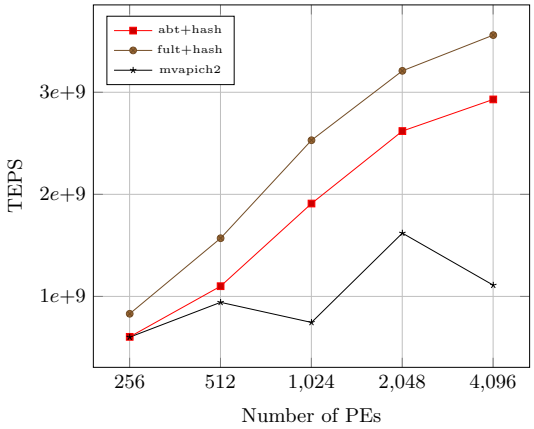
difficult to scale well due to the irregular access patterns and fine-grain communication. The benchmark provides four MPI reference BFS implementations. Among them, `graph500_bfs_simple` is a suitable candidate for us to re-implement since it uses MPI 2-sided point-to-point as the main communication method. Although this implementation has limited scalability, it is also simple to understand and is a frequent target of study. Jose et al. [27] point out that one of the main bottlenecks of this implementation is due to the Send/Recv communication model, which uses non-blocking communication to poll for arriving messages. We attempt to provide a simple remedy for this bottleneck by using blocking Send/Recv in combination with multi-threading. More specifically, we have made the following two important changes:

First, each MPI process traverses its assigned graph partition using multiple processing threads: Each vertex that is assigned to an MPI process is now further assigned to a thread spawned by that process. Each thread also maintains a separate traversing queue and appends to this queue when it traverses vertices that it owns, otherwise, it atomically appends to the queue of the owner threads. For this reason, in our modification, we use all the cores of a compute node within one MPI process, while the reference uses one MPI process per core.

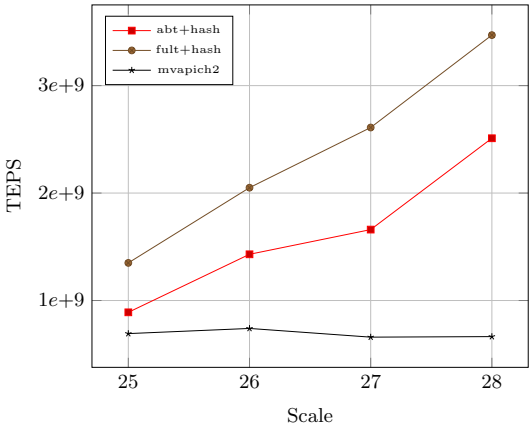
Second, when a vertex is owned by a remote process, the communication is done via blocking `MPI_SEND` and `MPI_RECV` instead of their non-blocking counterpart. A thread performs an `MPI_SEND` when it has accumulated enough vertices owned by the destination. The `MPI_RECV` is performed in a separate set of threads. These threads are spawned initially and only scheduled when a message has arrived. The woken up thread finds vertices that belong to the current MPI rank in the receive buffer, and appends them to the corresponding local thread queues; the appends are atomic. The non-blocking receive in the reference implementation uses `MPI_ANY_SOURCE`. Here we apply one of our mitigation strategies by having the number of receiving threads equal to the number of sending nodes.

Although this design could lead to a large number of threads, the receiving threads are not running when there is no incoming message, hence we do not waste CPU times as in the original algorithm. Moreover, our runtime is able to handle a very large number of threads efficiently as we have shown in the previous section. However, we admit that when the memory for storing thread records becomes the bottleneck, one must come up with a more sophisticated approach.

We compare our multi-threaded implementation with 15 workers and 1 communication server with the reference running 16 processes per node; other settings are kept as the default. The weak and strong scaling results of computed median TEPS are provided in Figure 8. We do not show the result for *pthread+hash* since the performance is far worse (10× slower than the reference). This is expected since the performance of our threaded version depends on the ability to context-switch efficiently between receiving threads and processing threads which happens very frequently in BFS due to the fine granularity of the communication. Our implementation using out ULT scheduler is able to scale BFS



(a) Strong scaling on 29-scale graph.



(b) Weak scaling with 512 PE per 25-scale graph (4096 cores at 28-scale)

Figure 8: Strong and Weak scaling for Graph500 in terms of number of Traversed Edges per Second (TEPS - the higher the better) for different MPI implementation.

to 4096 cores. At that scale, our Fult scheduler achieves 3× performance over the reference code and 20% better than Argobots.

4.3.4 Unbalanced Tree Search (UTS)

Unbalanced Tree Search is a benchmark for evaluating the performance of parallel systems under heavily unbalanced and irregular workloads [34]. The benchmark randomly generates a tree based on sampling from configured probability distribution and requires traversing every generated vertex. Unlike Graph500, an MPI work-stealing implementation is considered quite scalable and has been used for evaluating other runtime systems [11, 15, 33]. The basic idea is that an MPI process sends stealing requests to other MPI processes when it has explored all previously assigned vertices. All communications are done via non-blocking point-to-point MPI calls. The application is less communication bound than the Graph500 BFS, but requires dynamic coordination between the processes for balancing works.

We obtained the latest reference implementation (version 1.1) from the publicly available source at [3] and modified the work-stealing MPI code (`mpi_workstealing.c`) to match our MPI implementation. The modifications are in a similar style as those for Graph500: 1) Use multiple threads for each MPI process and allow each thread to explore in parallel multiple vertices in their own stack. When there are no more vertices to explore, the thread will first try to steal from other threads' stacks on the same node before trying to request work from a different MPI process. We make little effort to optimize the intra-process stealing and use locks to protect critical sections. 2) The reference implementation uses non-blocking communication to wait for work (incoming vertices or incoming stealing requests). Instead, we use multiple communicating threads. Each thread uses a blocking `MPI_RECV` and then acts upon the data it received.

We compare our multi-threaded implementation with 15 workers and 1 communication server with the reference running 16 processes per node; other settings are kept as the default. The strong scaling result is shown in Figure 9 for `T3XXL` tree (a 2.8 billion vertices, Binomial tree that is recommended by the package). Our implementation is 10× faster than the original code, 7× than with POSIX thread scheduler and 10% better than with Argobots at 4096 cores.

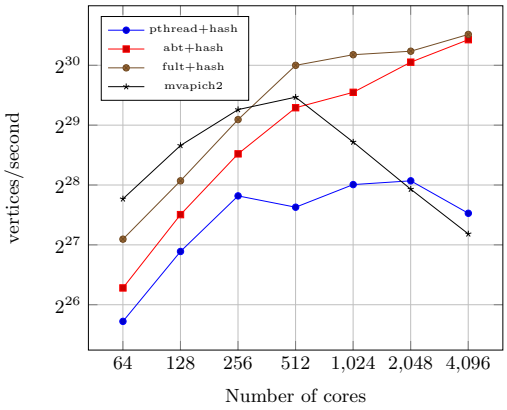


Figure 9: Strong scaling results for UTS of T3XXL tree in terms of vertices per second (the higher the better) under different MPI implementations.

5. RELATED WORKS

Although MPI is still the de-facto programming model in High Performance and Scientific Computing, there is a great deal of research in new programming models as we start to think about exascale. A common theme of these new models are the ability to support intra-node parallelism using multi-tasking or fork-join models [21], coupled with some form of distributed memory communication. For example, PPL—a recent programming model developed at University of Illinois—provides a message-driven, multi-threading runtime in distributed memory using one-sided communication and software cache [9]. A survey of modern programming models and features is also provided in the paper [9]. These programming models are alternative options for MPI, but our techniques of implementations are applicable to other programming models as well.

A common criticism of MPI is its inability to cope with increasing intra-node parallelism. Our work and the work of others show that the reason for these criticisms are less due to the message-passing model but rather to specific choices in current MPI definition and implementations.

There are several efforts to provide and improve multi-threading support in MPI; we name a few here. MIMPI [19] and MPICH-MT [41] are early designs and implementations of thread-safe MPI on distributed memory. Recent efforts from Dózsa [17] and Balaji [6] study the replacement of MPI coarse-grain lock by fine-grain locks and implement parallel receive queues using these locks; they demonstrate improvement in message rate up to 4 threads however it suffers mutex overheads and requires a complex, error-prone implementation (admitted by the authors). A more in-depth analysis of locking contention in MPI+Thread can be found in [4]. Recently, Intel researchers [18] also identified message matching as an important issue in MPI implementation and came up with a solution based on a hash-table. Their work was not focused on multi-threading, and they did not provide performance results for multi-threading. We shall be interested in comparing our work with theirs once their code becomes available.

A different aproach followed by many projects is to implement MPI processes as threads [25, 43, 12, 38, 28]. This largely solves the performance issues we discussed, since each thread has a diferent rank and can have its own private MPI data structures. On the other hand, this increases the number of ranks, increases memory usage, and does not support dynamic thread creation; it is a different programming model.

Off-loading MPI communication or network polling to dedicated cores/threads is another theme of research. This approach is often used when MPI communication is integrated into a light-weight threading runtime such as Habanero-C [11], or Qthreads [42]. Liu [31] demonstrates a general approach to incorporating user-level threading and MPI, giving different methods for network polling. We share the polling mechanism. However, these designs are based on top-down solutions: adding extra layers atop of MPI and therefore having higher latency. We believe that once MPI is aware of concurrent executions and designed with that in mind, many of these techniques will not be necessary. For example, in our design, there is no need for an efficient concurrent queue (which is more conflict-prone than a hash-table) for passing requests to the server (which is required by [11]), and there is no need for calling `MPI_PROBE` or `MPI_TEST` as [11, 42, 31].

It is worth mentioning that multi-threaded communication can be a solution for heavily communication-bound applications on multi-core clusters. In this approach, multiple threads or cores are cooperating to execute communication related codes. USFMPI [10], MT-MPI [40], and pioman [13] are a few that follow this direction. The technique is orthogonal to our work and is useful when more than one communication server is required to cope with a higher message injection rate. One must also watch out for performance degradation when there are too many concurrent messages in the current NIC architecture, as pointed out by Luo in [32]. Our packet pool design has already taken that into account.

6. DISCUSSION

The results in this paper raise two main questions:

1. Are the restrictions we impose on MPI reasonable; or do they prevent the use of important communication motifs?

2. Is it possible to relax the restrictions we imposed on MPI without losing the performance benefits of our approach?

We address each of these questions in turn.

6.1 MPI Restrictions

Wildcard receives are often used, with `MPI_ANY_SOURCE`, in order to handle messages from different sources in the order they arrive (thus coping with variable arrival times), or to support the one master, many slaves or one server, many clients motif in a scalable manner. The main problem in supporting wildcard receives is that they can match any send, so the matching of sends and receives cannot be split into non-interfering bins. The implementation problems would vanish if incoming messages carried an `MPI_ANY_SOURCE` identification – this essentially becomes a special source value that is used both by sender and by receiver, and matched using our current logic.

We conjecture that the code logic is such that it is seldom, if ever, the case that the same send operation is matched sometimes by a receive with a specific sender value and at other times by a receive with a wildcard sender value; the sender "knows" whether the matching receive will use a wildcard. If so, the proposed approach of "tagging" sent messages with an `MPI_ANY_SOURCE` tag matches currently used communication motifs.

The proposed approach can be implemented in MPI, for example, by having communicators that are always used with wildcard receives, or using a special tag value at the sender to indicate a wildcard receive.

The constraint of having each send possibly match only one receive and each receive possibly match only one send is less important and can be relaxed without significant changes in our implementation: We will need to ensure that concurrent accesses to the same bucket in the hash table traverse the entries in the bucket in order; an access either deletes the first matching entry or appends the new entry to to the end of the bucket list. Performance will degrade when many concurrent communications use the same `<communicator, sender, tag>` key, but we conjecture that such situations are rare and can (and should) be avoided.

We assumed that ULTs do not migrate once they started executing. ULT migration can be supported: It is essentially equivalent to deleting a bit from one bit vector (marking tha ULT as not runnable on the source worker); and inserting a bit into another bit vector (as done when a new ULT is spawned). The overhead occurs only when ULTs are migrated—presumably an operation that is less frequent and has a relatively large overhead.

6.2 Extensions

A full implementation of MPI message passing has to support datatypes; has to support multiple point-to-point communication modes; and has to handle a variety of special cases, such as specified by `MPI_STATUS_IGNORE` or `MPI_PROC_NULL`. Handling these issues increases the latency of executing sends and receives, but does not require changes in the basic design outlined in the paper, since it does not require additional coordination between threads; they will

not worsen the performance of `MPI_THREAD_MULTIPLE`, as compared to `MPI_THREAD_SINGLE`. A possible exception is the support of `MPI_WAITANY`, `MPI_WAITALL` and other similar calls: The use of such calls break the one-to-one relation between an incoming message and a waiting thread, and will require a more complex interaction between the communication server and the scheduler(s). Handling errors and edge cases like unmatched buffer size costs some cycles, they are seldom used but required for completeness. We intend to explore and evaluate these problems in future work.

Our tight integration between the thread scheduler and the communication layer may be problematic, in practice, since the thread scheduler needs to interact with other subsystems, such as the runtime of the language that is used for multithreading. The two can be separated—we only require an efficient implementation of `ThreadWait` and `ThreadSignal`. Our work indicates the need for standardizing an interface that supports such functionally with a very low overhead.

Another important extension is to support multiple communication servers in order to support higher message rates. As long as wildcards are not supported, this does not seem to raise major issues: For example, the hash table can be split into distinct hash tables, each handling a distinct range of key values; each communication server would be associated with a separate hash table. This, too, will be the subject of our future work.

6.3 Benchmarks

It would be very desirable to evaluate our design with full-fledged applications or, to the least, with representative mini-applications. Unfortunately such effort suffers from a "chicken and egg problem": Applications do not use `MPI_THREAD_MULTIPLE`, since implementations are inefficient; and implementations are not focused on `MPI_THREAD_MULTIPLE` since better implementations do not benefit and probably harm current application codes. We believe that `MPI_THREAD_FUNNELED` is not a sufficient answer for nodes with hundreds of concurrent hardware threads and for asynchronous task programming models. One important way of breaking this vicious cycle is to demonstrate the benefits of good `MPI_THREAD_MULTIPLE` support with mini-apps and full applications. We hope to be able to do so in the coming years.

7. CONCLUSION

It has become a common wisdom that using `MPI_THREAD_MULTIPLE` is a bad practice. In this paper, we have laid the foundation for overturning this statement, thus greatly extending the programmability and usability of MPI toward future architectures. We showed that using a more restrictive model of MPI send-receive, we can design a constant-time overhead message matching and delivery under highly concurrent executions without sacrificing performance of sequential usage.

Our method is a tight integration of communication layer, thread scheduler and resource management. Moreover, the entire protocol relies on only a handful of operations which could be potentially implemented in hardware. Using a variety of micro- and application- benchmarks, we proved the efficiency and performance benefit of this design. The implementation is able to maintain performance up to a million communicating threads and scale several applications which previously shown unscalable, using the same algorithm or with modest modifications.

Our future work will develop a complete MPI implementation on top of the existing protocol. While moving bottom-up, we shall evaluate and quantify the cost and benefit of generality vs. performance. We shall also study performance on systems with a large number of physical threads, such as the Xeon Phi. We believe the insight from this research will facilitate a discussion on how MPI should evolve in the upcoming era.

8. ACKNOWLEDGEMENTS

This material is based upon work supported by the U.S. Department of Energy, Office of Science, under contract number DE-AC02-06CH11357 and by a subcontract from Sandia National Laboratories. It was performed while the second author was at the MCS Division of the Argonne National Laboratory. We thank Ron Brightwell, Pavan Balaji, Nikoli Dryden and Alex Brooks for their help with this work.

9. REFERENCES

[1] Graph 500. http://www.graph500.org/. [Online; accessed 13-May-2016].

[2] TACC Stampede Cluster. http://www.xsede.org/resources/overview, 2016.

[3] The unbalanced tree search benchmark. https://sourceforge.net/projects/uts-benchmark/files/, 2016.

[4] A. Amer, H. Lu, Y. Wei, P. Balaji, and S. Matsuoka. MPI+threads: Runtime contention and remedies. *ACM SIGPLAN Notices*, 50(8):239–248, 2015.

[5] N. S. Arora, R. D. Blumofe, and C. G. Plaxton. Thread scheduling for multiprogrammed multiprocessors. *Theory of Computing Systems*, 34(2):115–144, 2001.

[6] P. Balaji, D. Buntinas, D. Goodell, W. Gropp, and R. Thakur. Toward efficient support for multithreaded MPI communication. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 120–129. Springer, 2008.

[7] P. Balaji, D. Buntinas, D. Goodell, W. Gropp, and R. Thakur. Fine-grained multithreading support for hybrid threaded MPI programming. *International Journal of High Performance Computing Applications*, 24(1):49–57, 2010.

[8] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[9] A. Brooks, H.-V. Dang, N. Dryden, and M. Snir. PPL: an abstract runtime system for hybrid parallel programming. In *Proceedings of the First International Workshop on Extreme Scale Programming Models and Middleware*, pages 2–9. ACM, 2015.

[10] S. G. Caglar, G. D. Benson, Q. Huang, and C.-W. Chu. USFMPI: a multi-threaded implementation of MPI for Linux clusters. In *Fifteenth IASTED International Conference on Parallel and Distributed Computing and Systems*, pages 674–680, 2003.

[11] S. Chatterjee, S. Tasırlar, Z. Budimlic, V. Cave, M. Chabbi, M. Grossman, V. Sarkar, and Y. Yan. Integrating asynchronous task parallelism with MPI. In *IEEE 27th International Symposium on Parallel &*

Distributed Processing (IPDPS), 2013, pages 712–725. IEEE, 2013.

[12] E. D. Demaine. A threads-only mpi implementation for the development of parallel programs. In *Proceedings of the 11th international symposium on high performance computing systems*, pages 153–163. Citeseer, 1997.

[13] A. Denis. pioman: a pthread-based multithreaded communication engine. In *Parallel, Distributed and Network-Based Processing (PDP), 2015 23rd Euromicro International Conference on*, pages 155–162. IEEE, 2015.

[14] P. Dhabaleswar. OSU Micro-Benchmarks 5.3. http://mvapich.cse.ohio-state.edu/benchmarks/, 2016. [Online; accessed 18-April-2016].

[15] J. Dinan, S. Olivier, G. Sabin, J. Prins, P. Sadayappan, and C.-W. Tseng. Dynamic load balancing of unbalanced computations using message passing. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1–8. IEEE, 2007.

[16] J. Dongarra, D. Walker, E. Lusk, B. Knighten, M. Snir, A. Geist, S. Otto, R. Hempel, E. Lusk, W. Gropp, et al. MPI: a message-passing interface standard. *International Journal of Supercomputer Applications and High Performance Computing*, 8(3-4):165, 1994.

[17] G. Dózsa, S. Kumar, P. Balaji, D. Buntinas, D. Goodell, W. Gropp, J. Ratterman, and R. Thakur. Enabling concurrent multithreaded MPI communication on multicore petascale systems. In *Recent Advances in the Message Passing Interface*, pages 11–20. Springer, 2010.

[18] M. Flajslik, J. Dinan, and K. D. Underwood. Mitigating MPI message matching misery. In *International Supercomputing Conference*, 2016.

[19] F. García, A. Calderón, and J. Carretero. Mimpi: A multithread-safe implementation of mpi. In *European Parallel Virtual Machine/Message Passing Interface UsersâĂŹ Group Meeting*, pages 207–214. Springer, 1999.

[20] A. Geist, W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, W. Saphir, T. Skjellum, and M. Snir. MPI-2: Extending the message-passing interface. In *Euro-Par'96 Parallel Processing*, pages 128–135. Springer, 1996.

[21] W. Gropp and M. Snir. Programming for exascale computers. *Computing in Science & Engineering*, 15(6):27–35, 2013.

[22] W. Gropp and R. Thakur. Issues in developing a thread-safe MPI implementation. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 12–21. Springer, 2006.

[23] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[24] C. A. R. Hoare. *Communicating sequential processes*. Springer, 1978.

[25] C. Huang, O. Lawlor, and L. V. Kale. Adaptive mpi. In *International workshop on languages and compilers for parallel computing*, pages 306–322. Springer, 2003.

[26] W. Huang, G. Santhanaraman, H.-W. Jin, Q. Gao, and D. K. Panda. Design of high performance MVAPICH2: MPI2 over InfiniBand. In *Sixth IEEE International Symposium on Cluster Computing and the Grid (CCGRID 06)*, volume 1, pages 43–48. IEEE, 2006.

[27] J. Jose, S. Potluri, K. Tomko, and D. K. Panda. Designing scalable Graph500 benchmark with hybrid MPI+OpenSHMEM programming models. In *Supercomputing*, pages 109–124. Springer, 2013.

[28] H. Kamal and A. Wagner. Fg-mpi: Fine-grain mpi for multicore and clusters. In *Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, pages 1–8. IEEE, 2010.

[29] X. Li, D. G. Andersen, M. Kaminsky, and M. J. Freedman. Algorithmic improvements for fast concurrent cuckoo hashing. In *Proceedings of the Ninth European Conference on Computer Systems*, page 27. ACM, 2014.

[30] J. Liu, J. Wu, and D. K. Panda. High performance RDMA-based MPI implementation over InfiniBand. *International Journal of Parallel Programming*, 32(3):167–198, 2004.

[31] H. Lu, S. Seo, and P. Balaji. MPI+ ULT: Overlapping communication and computation with user-level threads. In *High Performance Computing and Communications (HPCC), 2015 IEEE 17th International Conference on*, pages 444–454. IEEE, 2015.

[32] M. Luo, D. K. Panda, K. Z. Ibrahim, and C. Iancu. Congestion avoidance on manycore high performance computing systems. In *Proceedings of the 26th ACM international conference on Supercomputing*, pages 121–132. ACM, 2012.

[33] R. Machado, C. Lojewski, S. Abreu, and F.-J. Pfreundt. Unbalanced tree search on a manycore system using the GPI programming model. *Computer Science-Research and Development*, 26(3-4):229–236, 2011.

[34] S. Olivier, J. Huan, J. Liu, J. Prins, J. Dinan, P. Sadayappan, and C.-W. Tseng. UTS: An unbalanced tree search benchmark. In *Languages and Compilers for Parallel Computing*, pages 235–250. Springer, 2006.

[35] Message Passing Interface Forum. MPI 4.0 Standardization Effort, Point to Point Communication. https://svn.mpi-forum.org/trac/mpi-forum-web/wiki/PtpWikiPage. [Online; accessed 6-May-2016].

[36] NASA. NAS Parallel Benchmarks. http://www.nas.nasa.gov/publications/npb.html, 2016.

[37] C. Pheatt. Intel® threading building blocks. *Journal of Computing Sciences in Colleges*, 23(4):298–298, 2008.

[38] E. R. Rodrigues, P. O. A. Navaux, J. Panetta, and C. L. Mendes. A new technique for data privatization in user-level threads and its use in parallel applications. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, pages 2149–2154, New York, NY, USA, 2010. ACM.

[39] S. Seo, A. Amer, P. Balaji, P. Beckman, C. Bordage, G. Bosilca, A. Brooks, A. CastellÃş, D. Genet, T. Herault, P. Jindal, L. V. Kale, S. Krishnamoorthy, J. Lifflander, H. Lu, E. Meneses, M. Snir, and Y. Sun. Argobots: A lightweight low-level threading/tasking framework. Technical Report ANL/MCS-P5515-0116, 2016.

[40] M. Si, A. J. Peña, P. Balaji, M. Takagi, and Y. Ishikawa. MT-MPI: Multithreaded MPI for many-core environments. In *Proceedings of the 28th ACM international conference on Supercomputing*, pages 125–134. ACM, 2014.

[41] A. Skjellum, B. Protopopov, and S. Hebert. A thread taxonomy for mpi. In *MPI Developer's Conference, 1996. Proceedings., Second*, pages 50–57. IEEE, 1996.

[42] D. T. Stark, R. F. Barrett, R. E. Grant, S. L. Olivier, K. T. Pedretti, and C. T. Vaughan. Early experiences co-scheduling work and communication tasks for hybrid MPI+X applications. In *Proceedings of the 2014 Workshop on Exascale MPI*, pages 9–19. IEEE Press, 2014.

[43] H. Tang, K. Shen, and T. Yang. Compile/run-time support for threaded mpi execution on multiprogrammed shared memory machines. In *ACM SIGPLAN Notices*, volume 34, pages 107–118. ACM, 1999.

[44] The Open MPI Project. Is Open MPI thread safe. shttps://www.open-mpi.org/faq/?category=supported-systems#thread-support, 2016. [Online; accessed 8-May-2016].

This material is based upon work supported by the U.S. Department of Energy, Office of Science, under contract number DEAC0206CH11357.